

Mohammed Bekkouche H  l  ne Collavizza Michel Rueher

Résumé

Abstract

1 Introduction

La prochaine section est consacrée à un positionnement de notre approche par rapport aux principales méthodes qui ont été proposées pour ce résoudre ce problème. La section suivante est dédiée à la description de notre approche et des algorithmes utilisés. Puis, nous présentons les résultats des premières expérimentations avant de conclure.

*Ce travail a débuté au NII (National Institute of Informatics, 2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo 101-8430) où Michel Rueher a été invité plusieurs fois en tant que Professeur associé depuis 2011. Les idées générales et la démarche ont été définies lors des réunions de travail avec les professeurs Hiroshi HOSOE et Shin NAKAJIMA.

2 État de l'art

Dans cette section, nous positionnons notre approche par rapport aux principales méthodes existantes. Nous parlerons d'abord des méthodes utilisées pour l'aide à la localisation des erreurs dans le cadre du test et de la vérification de programmes. Comme l'approche que nous proposons consiste essentiellement à rechercher des sous-ensembles de contraintes spécifiques dans un système de contraintes inconsistent, nous parlerons aussi dans un second temps des algorithmes qui ont été utilisés en recherche opérationnelle et en programmation par contraintes pour aider l'utilisateur à debugger un système de contraintes inconsistent.

2.1 Méthodes d'aide à la localisation des erreurs utilisées en test et en vérification de programmes

Différentes approches ont été proposées pour aider le programmeur dans l'aide à la localisation d'erreurs dans la communauté test et vérification.

Le problème de la localisation des erreurs a d'abord été abordé dans le cadre du test où de nombreux systèmes ont été développés. Le plus célèbre est Tarantula [16, 15] qui utilise différentes métriques pour établir un classement des instructions suspectes détectées lors de l'exécution d'une batterie de tests. Le point critique de cette approche réside dans le fait qu'elle requiert un oracle qui permet de décider si le résultat du test est juste ou non. Nous nous plaçons ici dans un cadre moins exigeant (et plus réaliste) qui est celui du Bounded-Model Checking (BMC), c'est à dire un cadre où les seuls prérequis sont un programme, une post-condition ou une assertion qui doit être vérifiée, et éventuellement une pré-condition.

C'est aussi dans ce cadre que se placent Bal et al[1] qui utilisent plusieurs appels à un Model Checker et comparent les contre-exemples obtenus avec une trace d'exécution correcte. Les transitions qui ne figurent pas dans la trace correcte sont signalées comme une possible cause de l'erreur. Ils ont implanté leur algorithme dans le contexte de SLAM, un model checker qui vérifie les propriétés de sécurité temporelles de programmes C.

Plus récemment, des approches basées sur la dérivation de traces correctes ont été introduites dans un système nommé **Explain**[14, 13] et qui fonctionne en trois étapes :

1. Appel de **CBMC**¹ pour trouver une exécution qui viole la post-condition ;
2. Utilisation d'un solveur pseudo-booléen pour rechercher l'exécution correcte la plus proche ;
3. Calcul de la différence entre les traces.

Explain produit ensuite une formule propositionnelle S associée à P mais dont les affectations ne violent pas la spécification. Enfin **Explain** étend S avec des contraintes

représentant un problème d'optimisation : trouver une affectation satisfaisante qui soit aussi proche que possible du contre-exemple ; la proximité étant mesurée par une distance sur les exécutions de P .

Une approche similaire à **Explain** a été introduite dans [25] mais elle est basée sur le test plutôt que sur la vérification de modèles : les auteurs utilisent des séries de tests correctes et des séries erronées. Ils utilisent aussi des métriques de distance pour sélectionner un test correct à partir d'un ensemble donné de tests. Cette approche suppose qu'un oracle soit disponible.

Dans [11, 12], les auteurs partent aussi de la trace d'un contre-exemple, mais ils utilisent la spécification pour dériver un programme correct pour les mêmes données d'entrée. Chaque instruction identifiée est un candidat potentiel de faute et elle peut être utilisée pour corriger les erreurs. Cette approche garantit que les erreurs sont effectivement parmi les instructions identifiées (en supposant que l'erreur est dans le modèle erroné considéré). En d'autres termes, leur approche identifie un sur-ensemble des instructions erronées. Pour réduire le nombre d'erreurs potentielles, le processus est redémarré pour différents contre-exemples et les auteurs calculent l'intersection des ensembles d'instructions suspectes. Cependant, cette approche souffre de deux problèmes majeurs :

- Elle permet de modifier n'importe quelle expression, l'espace de recherche peut ainsi être très grand ;
- Elle peut renvoyer beaucoup de faux diagnostics totalement absurdes car toute modification d'expression est possible (par exemple, changer la dernière affectation d'une fonction pour renvoyer le résultat attendu).

Pour remédier à ces inconvénients, Zhang et al [28] proposent de modifier uniquement les prédicats de flux de contrôle. L'intuition de cette approche est qu'à travers un switch des résultats d'un prédicat et la modification du flot de contrôle, l'état de programme peut non seulement être modifié à peu de frais, mais qu'en plus, il est souvent possible d'arriver à un état succès. Liu et al [23] généralisent cette approche en permettant la modification de plusieurs prédicats. Ils proposent également une étude théorique d'un algorithme de débogage pour les erreurs *RHS*, c'est à dire les erreurs dans les prédicats de contrôle et la partie droite des affectations.

Dans [3], les auteurs abordent le problème de l'analyse de la trace d'un contre-exemple et de l'identification de l'erreur dans le cadre des systèmes de vérification formelle du hardware. Ils utilisent pour cela la notion de causalité introduite par Halpern et Pearl pour définir formellement une série de causes de la violation de la spécification par un contre-exemple.

Récemment, Manu Jose et Rupak Majumdar [17, 18] ont abordé ce problème différemment : ils ont introduit un nouvel algorithme qui utilise un solveur MAX-SAT pour calculer le nombre maximum des clauses d'une formule booléenne qui peut être satisfaite par une affecta-

1. <http://www.cprover.org/cbmc/>

tion. Leur algorithme fonctionne en trois étapes :

1. ils encodent une trace d'un programme par une formule booléenne F qui est satisfiable si et seulement si la trace est satisfiable ;
2. ils construisent une formule fausse F' en imposant que la post-condition soit vraie (la formule F' est insatisfiable car la trace correspond à un contre-exemple qui viole la post-condition) ;
3. Ils utilisent MAXSAT pour calculer le nombre maximum de clauses pouvant être satisfaites dans F' et affichent le complément de cet ensemble comme une cause potentielle des erreurs. En d'autres termes, ils calculent le complément d'un MSS (Maximal Satisfiable Subset).

Manu Jose et Rupak Majumdar [17, 18] ont implanté leur algorithme dans un outil appelé **BugAssist** qui utilise CBMC.

Si-Mohamed Lamraoui et Shin Nakajima [20] ont aussi développé récemment un outil nommé **SNIPER** qui calcule les MSS d'une formule $\psi = EI \wedge TF \wedge AS$ où EI encode les valeurs d'entrée erronées, TF est une formule qui représente tous les chemins du programme, et AS correspond à l'assertion qui est violée. Les MCS sont obtenus en prenant le complément des MSS calculés. L'implémentation est basée sur la représentation intermédiaire LLVM et le solveur SMT **Yices**. L'implémentation actuelle est toutefois beaucoup plus lente que **BugAssist**.

L'approche que nous proposons ici est inspirée des travaux de Manu Jose et Rupak Majumdar. Les principales différences sont :

1. Nous ne transformons pas tout le programme en un système de contraintes mais nous utilisons le graphe de flot de contrôle pour collecter les contraintes du chemin du contre exemple et des chemins dérivés de ce dernier en supposant qu'au plus k instructions conditionnelles sont susceptibles de contenir des erreurs.
2. Nous n'utilisons pas des algorithmes basés sur MAXSAT mais des algorithmes plus généraux qui permettent plus facilement de traiter des contraintes numériques.

2.2 Méthodes pour débayer un système de contraintes inconsistent

En recherche opérationnelle et en programmation par contraintes, différents algorithmes ont été proposés pour aider l'utilisateur à débayer un système de contraintes inconsistent. Lorsqu'on recherche des informations utiles pour la localisation des erreurs sur les systèmes de contraintes numériques, on peut s'intéresser à deux types d'informations :

1. Combien de contraintes dans un ensemble de contraintes insatisfiables peuvent être satisfaites ?

2. Où se situe le problème dans le système de contraintes ?

Avant de présenter rapidement les algorithmes² qui cherchent à répondre à ces questions, nous allons définir plus formellement les notions de MUS, MSS et MCS à l'aide des définitions introduites dans [22].

Un MUS (Minimal Unsatisfiable Subsets) est un ensemble de contraintes qui est inconsistent mais qui devient consistant si une contrainte quelconque est retirée de cet ensemble. Plus formellement, soit C un ensemble de contraintes :

$$M \subseteq C \text{ est un MUS} \Leftrightarrow M \text{ est UNSAT} \\ \text{et } \forall c \in M : M \setminus \{c\} \text{ est SAT.}$$

La notion de MSS (Maximal Satisfiable Subset) est une généralisation de MaxSAT / MaxCSP où l'on considère la maximalité au lieu de la cardinalité maximale :

$$M \subseteq C \text{ est un MSS} \Leftrightarrow M \text{ est SAT} \\ \text{et } \forall c \in C \setminus M : M \cup \{c\} \text{ est UNSAT.}$$

Cette définition est très proche de celle des IIS (Irreducible Inconsistent Subsystem) utilisés en recherche opérationnelle [5, 6, 7].

Les MCS (Minimal Correction Set) sont des compléments des MSS (le retrait d'un MCS à C produit un MSS car on "corrige" l'infaisabilité) :

$$M \subseteq C \text{ est un MCS} \Leftrightarrow C \setminus M \text{ est SAT} \\ \text{et } \forall c \in M : (C \setminus M) \cup \{c\} \text{ est UNSAT.}$$

Il existe donc une dualité entre l'ensemble des MUS et des MCS [4, 22] : informellement, l'ensemble des MCS est équivalent aux ensembles couvrants irréductibles³ des MUS ; et l'ensemble des MUS est équivalent aux ensembles couvrants irréductibles des MCS. Soit un ensemble de contraintes C :

1. Un sous-ensemble M de C est un MCS ssi M est un ensemble couvrant minimal des MUS de C ;
2. Un sous-ensemble M de C est un MUS ssi M est un ensemble couvrant minimal des MCS de C ;

Au niveau intuitif, il est aisé de comprendre qu'un MCS doit au moins retirer une contrainte de chaque MUS. Et comme un MUS peut être rendu satisfiable en retirant n'importe laquelle de ses contraintes, chaque MCS doit au moins contenir une contrainte de chaque MUS. Cette dualité est aussi intéressante pour notre problématique car elle montre que les réponses aux deux questions posées ci-dessus sont étroitement liées.

Différents algorithmes ont été proposés pour le calcul des IIS/MUS et MCS. Parmi les premiers travaux, on peut mentionner les algorithmes **Deletion Filter**, **Additive Method**, **Additive Deletion Method**, **Elastic Filter** qui ont été développés dans la communauté de recherche opérationnelle [5, 27, 6, 7]. Les trois premiers algorithmes sont des algorithmes itératifs alors que

2. Pour une présentation plus détaillée voir http://users.polytech.unice.fr/~rueher/Publis/Talk_NII_2013-11-06.pdf

3. Soit Σ un ensemble d'ensemble et D l'union des éléments de Σ . On rappelle que H est un ensemble couvrant de Σ si $H \subseteq D$ et $\forall S \in \Sigma : H \cup S \neq \emptyset$. H est irréductible (ou minimal) si aucun élément ne peut être retiré de H sans que celui-ci ne perde sa propriété d'ensemble couvrant.

le quatrième utilise des variables d'écart pour identifier dans la première phase du Simplexe les contraintes susceptibles de figurer dans un IIS.

Junker [19] a proposé un algorithme générique basé sur une stratégie "Divide-and-Conquer" pour calculer efficacement les IIS/MUS lorsque la taille des sous-ensembles conflictuels est beaucoup plus petite que celle de l'ensemble total des contraintes.

L'algorithme de Liffiton et Sakallah [22] qui calcule d'abord l'ensemble des MCS par ordre de taille croissante, puis l'ensemble des MUS est basé sur la propriété mentionnée ci-dessus. Cet algorithme, que nous avons utilisé dans notre implémentation est décrit dans la section suivante.

Différentes améliorations [10, 21, 24] de ces algorithmes ont été proposées ces dernières années mais elles sont assez étroitement liées à SAT et a priori assez difficilement transposables dans un contexte où nous avons de nombreuses contraintes numériques.

3 Notre approche

Dans cette section nous allons d'abord présenter le cadre général de notre approche, à savoir celui du "Bounded Model Checking" (BMC) basé sur la programmation par contraintes, puis nous allons décrire la méthode proposée et les algorithmes utilisés pour calculer des MCS de cardinalité bornée.

3.1 Les principes : BMC et MCS

Notre approche se place dans le cadre du "Bounded model Checking" (BMC) par programmation par contraintes [8, 9]. En BMC, les programmes sont dépliés en utilisant une borne b , c'est à dire que les boucles sont remplacées par des imbrications de conditionnelles de profondeur au plus b . Il s'agit ensuite de détecter des non-conformités par rapport à une spécification. Étant donné un triplet de Hoare $\{PRE, PROG_b, POST\}$, où PRE est la pré-condition, $PROG_b$ est le programme déplié b fois et $POST$ est la post-condition, le programme est *non conforme* si la formule $\Phi = PRE \wedge PROG_b \wedge \neg POST$ est satisfiable. Dans ce cas, une instantiation des variables de Φ est un *contre-exemple*, et un cas de non conformité, puisqu'il satisfait à la fois la pré-condition et le programme, mais ne satisfait pas la post-condition.

CPBPV [8] est un outil de BMC basé sur la programmation par contraintes. CPBPV transforme PRE et $POST$ en contraintes, et transforme $PROG_b$ en un CFG dans lequel les conditions et les affectations sont traduites en contraintes⁴. CPBPV construit le CSP de la formule Φ à la volée, par un parcours en profondeur du graphe. À l'état initial, le CSP contient les contraintes de PRE et $\neg POST$, puis les contraintes d'un chemin

sont ajoutées au fur et à mesure de l'exploration du graphe. Quand le dernier noeud d'un chemin est atteint, la faisabilité du CSP est testée. S'il est consistant, alors on a trouvé un contre-exemple, sinon, un retour arrière est effectué pour explorer une autre branche du CFG. Si tous les chemins ont été explorés sans trouver de contre-exemple, alors le programme est conforme à sa spécification (sous réserve de l'hypothèse de dépliage des boucles).

Les travaux présentés dans cet article cherchent à *localiser* l'erreur détectée par la phase de BMC. Plus précisément, soit CE une instantiation des variables qui satisfait le CSP contenant les contraintes de PRE et $\neg POST$, et les contraintes d'un chemin incorrect de $PROG_b$ noté $PATH$. Alors le CSP $C = CE \cup PRE \cup PATH \cup POST$ est *inconsistant*, puisque CE est un contre-exemple et ne satisfait donc pas la post-condition. Un *ensemble minima de correction* (ou MCS - Minimal Correction Set) de C est un ensemble de contraintes qu'il faut nécessairement enlever pour que C devienne consistant. Un tel ensemble fournit donc une *localisation de l'erreur* sur le chemin du contre-exemple. Comme l'erreur peut se trouver dans une affectation sur le chemin du contre-exemple, mais peut aussi provenir d'un mauvais branchement, notre approche (nommée **LocFaults**) s'intéresse également aux MCS des systèmes de contraintes obtenus en déviant des branchements par rapport au comportement induit par le contre-exemple. Plus précisément, l'algorithme *LocFaults* effectue un parcours en profondeur d'abord du CFG de $PROG_b$, en propageant le contre-exemple et en déviant au plus k_{max} conditions. Trois cas peuvent être distingués :

- Aucune condition n'a été déviée : **LocFaults** a parcouru le chemin du contre-exemple en collectant les contraintes de ce chemin et il va calculer les MCS sur cet ensemble de contraintes ;
- k_{max} conditions ont été déviées sans qu'on arrive à trouver un chemin qui satisfasse la post-condition : on abandonne l'exploration de ce chemin ;
- d conditions ont déjà été déviées et on peut encore dévier au moins une condition, c'est à dire $k_{max} > 1$. Alors la condition courante c est déviée. Si le chemin résultant ne viole plus la post-condition, l'erreur sur le chemin initial peut avoir deux causes différentes :
 - (i) les conditions déviées elles-mêmes sont cause de l'erreur,
 - (ii) une erreur dans une affectation a provoqué une mauvaise évaluation de c , faisant prendre le mauvais branchement.

Dans le cas (ii), le CSP $CE \cup PRE \cup PATH_c \cup \{c\}$, où $PATH_c$ est l'ensemble des contraintes du chemin courant, c'est à dire le chemin du contre-exemple dans lequel d déviations ont déjà été prises, est satisfiable. Par conséquent, le CSP $CE \cup PRE \cup PATH_c \cup \{\neg c\}$ est *insatisfiable*. **LocFaults** calcule donc également les MCS de ce CSP, afin de détecter les instructions suspectées d'avoir induit le mauvais bran-

4. Pour éviter les problèmes de re-définitions multiples des variables, la forme DSA (Dynamic Single Assignment [2]) est utilisée

Algorithm 3: correct

```

1 Fonction correct( $n, P$ )
  Entrées:
  –  $n$  : noeud du CFG,
  –  $P$  : contraintes de propagation (issues du contre-exemple et du
    chemin),
  Sorties: true si le programme est correct sur le chemin
    induit par  $P$ 
2 début
3   si  $n$  est la postcondition alors
4     si  $P \cup \{cstr(POST)\}$  est faisable alors
5       retourner true
6     fin
7     sinon
8       retourner false
9     fin
10  fin
11  sinon si  $n$  est un noeud conditionnel alors
12    si  $P \cup \{cstr(n.cond)\}$  est faisable alors
13      % exploration de la branche If
14      retourner correct( $n.left, P$ )
15    fin
16    sinon
17      retourner correct( $n.right, P$ )
18    fin
19  fin
20  sinon si ( $n$  est un bloc d'affectations) alors
21    % on propage les affectations
22    pour chaque affectation  $ass \in n.assigns$  faire
23       $P.add(propagate(ass, P))$ 
24    fin
25    % On continue l'exploration sur le noeud suivant
26    retourner
27    correct( $n.next, P, CSP_d, CSP_a, MCS, MCS_b$ )
28 fin

```

chement pour c .

Bien entendu, nous ne calculons pas les MCS des chemins ayant le même préfixe : si la déviation d'une condition permet de satisfaire la post-condition, il est inutile de chercher à modifier des conditions supplémentaires dans la suite du chemin

3.2 Description de l'algorithme

L'algorithme **LocFaults** (cf. Algorithm 1) prend en entrée un programme déplié non conforme vis-à-vis de sa spécification, un contre-exemple, et une borne maximum de la taille des ensembles de correction. Il dévie au plus k_{max} conditions par rapport au contre-exemple fourni, et renvoie une liste de corrections possibles.

LocFaults commence par construire le CFG du programme puis appelle la fonction **DFS** sur le chemin du contre-exemple (i.e. en déviant 0 condition) puis en acceptant au plus k_{max} déviations. La fonction **DFS** gère trois ensembles de contraintes :

- CSP_d : l'ensemble des contraintes des conditions qui ont été déviées à partir du chemin du contre-exemple,
- CSP_a : l'ensemble des contraintes d'affectations du chemin,
- P : l'ensemble des contraintes dites de propagation, c'est à dire les contraintes de la forme *variable* = *constante* qui sont obtenues en propageant le contre exemple sur les affectations du chemin.

Algorithm 4: MCS

```

1 Fonction MCS( $C, MCS_b$ )
  Entrées:  $C$  : Ensemble de contraintes infaisable,
   $MCS_b$  : Entier
  Sorties:  $MCS$  : Liste de MCS de  $C$  de cardinalité
    inférieure à  $MCS_b$ 
2 début
3    $C' \leftarrow ADDYVARS(C)$ 
4    $MCS \leftarrow \emptyset$ 
5    $k \leftarrow 1$ 
6   tant que  $SAT(C') \wedge k \leq MCS_b$  faire
7      $C'_k \leftarrow C' \wedge ATMOST(\{\neg y_1, \neg y_2, \dots, \neg y_n\}, k)$ 
8     tant que  $SAT(C'_k)$  faire
9        $MCS.add(newMCS)$ .
10     $C'_k \leftarrow C'_k \wedge$ 
11    BLOCKINGCLAUSE( $newMCS$ )
12     $C' \leftarrow C' \wedge$ 
13    BLOCKINGCLAUSE( $newMCS$ ).
14  fin
15   $k \leftarrow k + 1$ .
16 fin

```

L'ensemble P est utilisé pour propager les informations et vérifier si une condition est satisfaite, les ensembles CSP_d et CSP_a sont utilisés pour calculer les MCS . Ces trois ensembles sont collectés à la volée lors du parcours en profondeur. Les paramètres de la fonction **DFS** sont les ensembles CSP_d , CSP_a et P décrits ci-dessus, n le noeud courant du CFG, MCS la liste des corrections en cours de construction, k le nombre de déviations autorisées et MCS_b la borne de la taille des MCS . Nous notons $n.left$ (resp. $n.right$) la branche *if* (resp. *else*) d'un noeud conditionnel, et $n.next$ le noeud qui suit un bloc d'affectation ; $cstr$ est la fonction qui traduit une condition ou affectation en contraintes.

Le parcours commence avec CSP_d et CSP_a vides et P contenant les contraintes du contre-exemple. Il part de la racine de l'arbre ($CFG.root$) qui contient la pré-condition, et se déroule comme suit :

- Quand le dernier noeud est atteint (i.e. noeud de la post-condition), on est sur le chemin du contre-exemple. La post-condition est ajoutée à CSP_a et on cherche les MCS ,
- Quand le noeud est un noeud conditionnel, alors on utilise P pour savoir si la condition est satisfaite. Si on peut encore prendre une déviation (i.e. $k > 0$), on essaie de dévier la condition courante c et on vérifie si cette déviation corrige le programme en appelant la fonction *correct*. Cette fonction propage tout simplement le contre-exemple sur le graphe à partir du noeud courant et renvoie vrai si le programme satisfait la postcondition pour ce chemin,
- Si dévier c a corrigé le programme, alors les conditions qui ont été déviées (i.e. $CSP_d \cup c$) sont des corrections. De plus, on calcule aussi les correc-

Algorithm 1: LocFaults

```
1 Fonction LocFaults( $PROG_b, CE, k_{max}, MCS_b$ )
  Entrées:
  -  $PROG_b$  : un programme déplié  $b$  fois non conforme vis-à-vis de sa spécification,
  -  $CE$  : un contre-exemple de  $PROG_b$ ,
  -  $k_{max}$  : le nombre maximum de conditions à dévier,
  -  $MCS_b$  : la borne du cardinal des MCS
  Sorties: une liste de corrections possibles
2 début
3    $CFG \leftarrow CFG\_build(PROG_b)$  % construction du CFG
4    $MCS = []$ 
5    $DFS_{devie}(CFG.root, CE, \emptyset, \emptyset, 0, MCS, MCS_b)$  % calcul des MCS sur le chemin du contre-exemple
6    $DFS_{devie}(CFG.root, CE, \emptyset, \emptyset, k_{max}, MCS, MCS_b)$  % calcul des MCS en prenant au plus  $k_{max}$  déviations
7   retourner  $MCS$ 
8 fin
```

Algorithm 2: DFS_{devie}

```
1 Fonction  $DFS_{devie}(n, P, CSP_d, CSP_a, k, MCS, MCS_b)$ 
  Entrées:
  -  $n$  : noeud du CFG,
  -  $P$  : contraintes de propagation (issues du contre-exemple et du chemin),
  -  $CSP_d$  : contraintes des conditions déviées,
  -  $CSP_a$  : contraintes des affectations,
  -  $k$  : nombre de conditions à dévier,
  -  $MCS$  : ensemble des MCS calculés,
  -  $MCS_b$  : la borne du cardinal des MCS
2 début
3   si  $n$  est la postcondition alors
4     % on est sur le chemin du CE, calcul des MCS
5      $CSP_a \leftarrow CSP_a \cup \{cstr(POST)\}$ 
6      $MCS.add(MCS(CSP_a, MCS_b))$ 
7   fin
8   sinon si  $n$  est un noeud conditionnel alors
9     si  $P \cup \{cstr(n.cond)\}$  est faisable alors
10      % next est le noeud où l'on doit aller, devie est la branche opposée
11       $next = n.gauche$ 
12       $devie = n.droite$ 
13    fin
14    sinon
15       $next = n.droite$ 
16       $devie = n.gauche$ 
17    fin
18    si  $k > 0$  alors
19      % on essaie de dévier la condition courante
20       $corrige = correct(devie, P)$ 
21      si corrige alors
22        % le chemin est corrigé, on met à jour les MCS
23         $CSP_d \leftarrow CSP_d \cup \{cstr(n.cond)\}$ 
24         $MCS.addAll(CSP_d)$  % ajout des conditions déviées
25        % calcul des MCS sur le chemin qui mène à la dernière condition déviée
26        pour chaque  $c$  dans  $CSP_d$  faire
27           $CSP_a \leftarrow CSP_a \cup \{\neg c\}$ 
28        fin
29         $MCS.add(MCS(CSP_a, MCS_b))$ 
30      fin
31      sinon si  $k > 1$  alors
32        % on essaie de dévier la condition courante et des conditions en dessous
33         $DFS_{devie}(devie, P, CSP_d \cup \{cstr(n.cond)\}, CSP_a, k - 1, MCS, MCS_b)$ 
34      fin
35      % dans tous les cas, on essaie de dévier les conditions en dessous du noeud courant
36       $DFS_{devie}(next, P, CSP_d, CSP_a, k, MCS, MCS_b)$ 
37    fin
38    sinon
39      %  $k=0$ , on est sur le chemin du contre-exemple, on suit le chemin
40       $DFS_{devie}(next, P, CSP_d, CSP_a, k, MCS, MCS_b)$ 
41    fin
42  fin
43  sinon si ( $n$  est un bloc d'affectations) alors
44    pour chaque affectation  $ass \in n.assigns$  faire
45       $P.add(propagate(ass, P))$ 
46       $CSP_a \leftarrow CSP_a \cup \{cstr(ass)\}$ 
47    fin
48    % On continue l'exploration sur le noeud suivant
49     $DFS_{devie}(n.next, CE, P, CSP_d, CSP_a, k, MCS, MCS_b)$ 
50  fin
51 fin
```

tions dans le chemin menant à c ,

- Si dévier c n’a pas corrigé le programme, si on peut encore dévier des conditions (i.e. $k \geq 1$) alors on dévie c et on essaie de dévier $k - 1$ conditions en dessous de c .

Dans les deux cas (dévier c a corrigé ou non le programme), on essaie aussi de dévier des conditions en dessous de c , sans dévier c .

- Quand le noeud est un bloc d’affectations, on propage le contre-exemple sur ces affectations et on ajoute les contraintes correspondantes dans P et dans CSP_a .

L’algorithme **LocFaults** appelle l’algorithme **MCS** (cf. Algorithm 4) qui est une transcription directe de l’algorithme proposé par Liffiton et Sakallah [22]. Cet algorithme associe à chaque contrainte un sélecteur de variable y_i qui peut prendre la valeur 0 ou 1 ; la contrainte **AtMost** permet donc de retenir au plus k contraintes du système de contraintes initial dans le MCS. La procédure **BlockingClause(newMCS)** appelée à la ligne 10 (resp. ligne 11) permet d’exclure les sur-ensembles de taille k (resp. de taille supérieure à k).

Lors de l’implémentation de cet algorithme nous avons utilisé IBM ILOG CPLEX⁵ qui permet à la fois une implémentation aisée de la fonction **AtMost** et la résolution de systèmes de contraintes numériques. Il faut toutefois noter que cette résolution n’est correcte que sur les entiers et que la prise en compte des nombres flottants nécessite l’utilisation d’un solveur spécialisé pour le traitement des flottants.

4 Evaluation expérimentale

Pour évaluer la méthode que nous avons proposée, nous avons comparé les performances de **LocFaults** et de **BugAssist** [17, 18] sur un ensemble de programmes. Comme **LocFaults** est basé sur **CPBPV**[8] qui travaille sur des programmes Java et que **BugAssist** travaille sur des programmes C, nous avons construit pour chacun des programmes :

- une version en Java annotée par une spécification JML ;
- une version en ANSI-C annotée par la même spécification mais en ACSL.

Les deux versions ont les mêmes numéros de ligne et les mêmes instructions. La précondition est un contre-exemple du programme, et la postcondition correspond au résultat de la version correcte du programme pour les données du contre-exemple. Nous avons considéré qu’au plus trois conditions pouvaient être fausses sur un chemin. Par ailleurs, nous n’avons pas cherché de MCS de cardinalité supérieure à 3. Les expérimentations ont été effectuées avec un processeur Intel Core i7-3720QM 2.60 GHz avec 8 GO de RAM.

Nous avons d’abord utilisé un ensemble de programmes académiques de petite taille (entre 15 et 100 lignes). A savoir :

- **AbsMinus**. Ce programme prend en entrée deux entiers i et j et renvoie la valeur absolue de $i - j$.
- **Minmax**. Ce programme prend en entrée trois entiers : $in1$, $in2$ et $in3$, et permet d’affecter la plus petite valeur à la variable *least* et la plus grande valeur à la variable *most*.
- **Tritype**. Ce programme est un programme classique qui a été utilisé très souvent en test et vérification de programmes. Il prend en entrée trois entiers (les côtés d’un triangle) et retourne 3 si les entrées correspondent à un triangle équilatéral, 2 si elles correspondent à un triangle isocèle, 1 si elles correspondent à un autre type de triangle, 4 si elles ne correspondent pas à un triangle valide.
- **TriPerimetre**. Ce programme a exactement la même structure de contrôle que *tritype*. La différence est que *TriPerimetre* renvoie la somme des côtés du triangle si les entrées correspondent à un triangle valide, et -1 dans le cas inverse.

Pour chacun de ces programmes, nous avons considéré différentes versions erronées.

Nous avons aussi évalué notre approche sur les programmes TCAS (Traffic Collision Avoidance System) de la suite de test Siemens[26]. Il s’agit là aussi d’un benchmark bien connu qui correspond à un système d’alerte de trafic et d’évitement de collisions aériennes. Il y a 41 versions erronées et 1608 cas de tests. Nous avons utilisé toutes les versions erronées sauf celles dont l’indice *AltLayerValue* déborde du tableau *PositiveRAAltThresh* car les débordements de tableau ne sont pas traités dans **CPBPV**. A savoir, les versions **TcasKO...TcasKO41**. Les erreurs dans ces programmes sont provoquées dans des endroits différents. 1608 cas de tests sont proposés, chacun correspondant à contre-exemple. Pour chacun de ces cas de test T_j , on construit un programme $TcasV_iT_j$ qui prend comme entrée le contre-exemple, et dont postcondition correspond à la sortie correcte attendue.

Le code source de l’ensemble des programmes est disponible à l’adresse http://www.i3s.unice.fr/~bekkouch/Bench_Mohammed.html.

La table 1 contient les résultats pour le premier ensemble de programmes :

- Pour **LocFaults** nous affichons la liste des MCS. La première ligne correspond aux MCS identifiés sur le chemin initial. Les lignes suivantes aux MCS identifiés sur les chemins pour lesquels la postcondition est satisfaite lorsqu’une condition est déviée. Le numéro de la ligne correspondant à la condition est souligné.
- Pour **BugAssist** les résultats correspondent à la fusion de l’ensemble des compléments des MSS calculés, fusion qui est opérée par **BugAssist** avant l’affichage des résultats.

5. <http://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/>

Programme	Contre-exemple	Erreurs	LocFaults				BugAssist					
			= 0	≤ 1	≤ 2	≤ 3						
AbsMinusKO	$\{i = 0, j = 1\}$	17	{17}	{17}	{17}	{17}	{17}					
AbsMinusKO2	$\{i = 0, j = 1\}$	11	{11}, {17}	{11}, {17}	{11}, {17}	{11}, {17}	{17, 20, 16}					
AbsMinusKO3	$\{i = 0, j = 1\}$	14	{20}	{16}, {14}, {12}, {20}	{16}, {14}, {12}, {20}	{16}, {14}, {12}, {20}	{16, 20}					
MinmaxKO	$\{in_1 = 2, in_2 = 1, in_3 = 3\}$	19	{10}, {19}	{18}, {10}, {10}, {19}	{18}, {10}, {10}, {19}	{18}, {10}, {10}, {19}	{18, 19, 22}					
MidKO	$\{a = 2, b = 1, c = 3\}$	19	{19}	{19}	{19}	{14, 23, 26}, {19}	{14, 19, 30}					
Maxmin6varKO	$\{a = 1, b = -4, c = -3, d = -1, e = 0, f = -4\}$	27	{28}	{15}, {27}, {28}	{15}, {27}, {28}	{15}, {27}, {28}	{15, 12, 27, 31, 166}					
Maxmin6varKO2	$\{a = 1, b = -3, c = 0, d = -2, e = -1, f = -2\}$	12	{65}	{12}, {65}	{12}, {65}	{12}, {65}	{12, 64, 166}					
Maxmin6varKO3	$\{a = 1, b = -3, c = 0, d = -2, e = -1, f = -2\}$	12, 15	{65}	{65}	{12, 15}, {65}	{12, 15}, {65}	{12, 15, 64, 166}					
Maxmin6varKO4	$\{a = 1, b = -3, c = -4, d = -2, e = -1, f = -2\}$	12, 15, 19	{116}	{116}	{116}	{12, 15, 19}, {116}	{12, 166}					
TritypeKO	$\{i = 2, j = 3, k = 2\}$	54	{54}	{26}	{26}	{26}	{26, 27, 32, 33, 36, 48, 57, 68}					
				{48}, {30}, {25}	{29, 32}	{29, 32}						
				{54}	{48}, {30}, {25}, {53, 57}, {25}, {30}	{29, 35, 57}, {25}, {32, 44, 57}, {33}, {25}, {30}						
					{54}	{48}, {30}, {25}, {53, 57}, {25}, {30}						
						{54}		{54}				
						TritypeKO2		$\{i = 2, j = 2, k = 4\}$	53	{54}	{21}	{21}
{26}	{26}	{26}										
{35}, {27}, {25}, {53}, {25}, {27}	{29, 57}, {30}, {27}, {25}	{29, 57}, {30}, {27}, {25}, {32, 44}, {33}, {25}, {27}										
{54}	{32, 44}, {33}, {25}, {27}	{35}, {27}, {25}, {53}, {25}, {27}										
	{54}	{54}	{54}									
		{54}	{54}									
TritypeKO2V2	$\{i = 1, j = 2, k = 1\}$	31	{50}	{21}	{21}	{21}	{21, 26, 27, 29, 31, 33, 34, 36, 37, 49, 68}					
				{26}	{26}	{26}						
				{29}	{29}	{29}						
				{36}, {31}, {25}, {49}, {31}, {25}	{33, 45}, {34}, {31}, {25}	{33, 45}, {34}, {31}, {25}, {36}, {31}, {25}, {49}, {31}, {25}						
				{50}	{50}	{50}						
					{50}	{50}						
TritypeKO3	$\{i = 1, j = 2, k = 1\}$	53	{54}	{21}	{21}	{21}	{21, 26, 27, 29, 30, 32, 33, 35, 36, 48, 53, 68}					
				{29}	{26, 57}, {30}, {25}, {27}	{26, 57}, {30}, {25}, {27}						
				{35}, {30}, {25}, {53}, {30}, {25}	{27}	{27}						
				{54}	{29}	{29}						
					{54}	{32, 44}, {33}, {30}, {25}		{32, 44}, {33}, {30}, {25}				
						{35}, {30}, {25}, {53}, {30}, {25}		{35}, {30}, {25}, {53}, {30}, {25}				
TritypeKO4	$\{i = 2, j = 3, k = 3\}$	45	{46}	{45}, {33}, {25}	{26, 32}, {29, 32}	{26, 32}, {29, 32}	{26, 27, 29, 30, 32, 33, 35, 45, 49, 68}					
				{46}	{45}, {33}, {25}	{32, 35, 49}, {25}, {32, 35, 53}, {25}, {32, 35, 57}, {25}						
					{46}	{45}, {33}, {25}		{45}, {33}, {25}				
						{46}		{46}				
					TritypeKO5	$\{i = 2, j = 3, k = 3\}$		32, 45	{40}	{26}	{26}	{26}
				{29}						{29}	{29}	
{40}	{32, 45}, {33}, {25}	{32, 45}, {33}, {25}										
	{40}	{35, 49}, {25}, {35, 53}, {25}, {35, 57}, {25}	{35, 49}, {25}, {35, 53}, {25}, {35, 57}, {25}									
		{40}	{40}	{40}								
TritypeKO6	$\{i = 2, j = 3, k = 3\}$	32, 33	{40}	{26}			{26}			{26}	{26, 27, 29, 30, 32, 33, 35, 49, 68}	
				{29}	{29}	{29}						
				{40}	{35, 49}, {25}, {35, 53}, {25}, {35, 57}, {25}	{32, 45, 49}, {33}, {25}, {32, 45, 53}, {33}, {25}, {32, 45, 57}, {33}, {25}						
					{40}	{35, 49}, {25}, {35, 53}, {25}, {35, 57}, {25}	{35, 49}, {25}, {35, 53}, {25}, {35, 57}, {25}					
						{40}	{40}	{40}				
				TriPerimetreKO	$\{i = 2, j = 1, k = 2\}$	58	{58}	{31}	{31}	{31}		{28, 29, 31, 32, 35, 37, 65, 72}
{37}, {32}, {27}, {58}	{37}, {32}, {27}, {58}	{37}, {32}, {27}, {58}										
{32}	{32}	{32}										
TriPerimetreKOV2	$\{i = 2, j = 3, k = 2\}$	34	{60}, {34}	{40}, {33}, {27}	{40}, {33}, {27}	{40}, {33}, {27}	{28, 32, 33, 34, 36, 38, 40, 41, 52, 55, 56, 60, 64, 67, 74}					
				{60}, {34}	{60}, {34}	{60}, {34}						

TABLE 1 – MCS identifiés par LocFaults pour des programmes sans boucles

Sur ces benchmarks les résultats de LocFaults sont plus concis et plus précis que ceux de BugAssist.

La table 2 fournit les temps de calcul : dans les deux cas, P correspond au temps de prétraitement et L au temps de calcul des MCS. Pour LocFaults, le temps de pré-traitement inclut la traduction du programme Java en un arbre de syntaxe abstraite avec l'outil JDT (Eclipse Java development tools), ainsi que la construction du CFG dont les noeuds sont des ensembles de contraintes. C'est la traduction Java qui est la plus longue. Pour BugAssist, le temps de prétraitement est celui la construction de la formule SAT. Globalement,

les performances de LocFaults et BugAssist sont similaires bien que le processus d'évaluation de nos systèmes de contraintes soit loin d'être optimisé.

La table 3 donne les résultats pour les programmes de la suite TCAS. La colonne Nb_E indique pour chaque programme le nombre d'erreurs qui ont été introduites dans le programme alors que la colonne Nb_CE donne le nombre de contre-exemples. Les colonnes LF et BA indiquent respectivement le nombre de contre-exemples pour lesquels LocFaults et BugAssist ont identifié l'instruction erronée. On remarquera que LocFaults se compare favorablement à BugAssist sur ce benchmark

Programme	LocFaults					BugAssist	
	P	L				P	L
		= 0	≤ 1	≤ 2	≤ 3		
AbsMinusKO	0,487s	0,044s	0,073s	0,074s	0,062s	0,02s	0,03s
AbsMinusKO2	0,484s	0,085s	0,065s	0,085s	0,078s	0,01s	0,06s
AbsMinusKO3	0,479s	0,076s	0,113s	0,357s	0,336s	0,02s	0,04s
MinmaxKO	0,528s	0,243s	0,318s	0,965s	1,016s	0,01s	0,09s
MidKO	0,524s	0,065s	0,078s	0,052s	0,329s	0,02s	0,08s
Maxmin6varKO	0,528s	0,082s	0,132s	0,16s	0,149s	0,06s	1,07s
Maxmin6varKO2	0,536s	0,064s	0,072s	0,097s	0,126s	0,06s	0,66s
Maxmin6varKO3	0,545s	0,066s	0,061s	0,29s	0,307s	0,04s	1,19s
Maxmin6varKO4	0,538s	0,06s	0,07s	0,075s	0,56s	0,04s	0,78s
TritypeKO	0,493s	0,022s	0,097s	0,276s	2,139s	0,03s	0,35s
TritypeKO2	0,51s	0,023s	0,25s	2,083	3,864s	0,02s	0,69s
TritypeKO2V2	0,514s	0,034s	0,28s	1,178s	1,31s	0,02s	0,77s
TritypeKO3	0,493s	0,022s	0,26s	1,928s	4,535s	0,02	0,48s
TritypeKO4	0,497s	0,023s	0,095s	0,295	5,127s	0,02s	0,21s
TritypeKO5	0,492s	0,021s	0,099s	0,787s	0,8s	0,01s	0,25s
TritypeKO6	0,492s	0,025s	0,078s	0,283s	1,841s	0,03s	0,24s
TriPerimetreKO	0,518s	0,047s	0,126s	1,096s	2,389s	0,03s	0,64s
TriPerimetreKOV2	0,503s	0,043s	0,271s	0,639s	1,958s	0,03s	1,20s

TABLE 2 – Temps de calcul

qui ne contient quasiment aucune instruction arithmétique; comme précédemment le nombre d'instructions suspectes identifiées par **LocFaults** est dans l'ensemble nettement inférieur à celui de **BugAssist**.

Les temps de calcul de **BugAssist** et **LocFaults** sont très similaires et inférieurs à une seconde pour chacun des benchmarks de la suite TCAS.

5 Discussion

Nous avons présenté dans cet article une nouvelle approche pour l'aide à la localisation d'erreurs qui utilise quelques spécificités de la programmation par contraintes. Les premiers résultats sont encourageants mais doivent encore être confirmés sur des programmes plus importants et contenant plus d'opérations arithmétiques.

Au niveau des résultats obtenus **LocFaults** est plus précis que **BugAssist** lorsque les erreurs sont sur le chemin du contre exemple ou dans une des conditions du chemin du contre-exemple. Ceci provient du fait que **BugAssist** et **LocFaults** ne calculent pas exactement la même chose :

BugAssist calcule les compléments des différents sous-ensembles obtenus par **MaxSat**, c'est à dire des sous-ensembles de clauses satisfiables de cardinalité maximale. Certaines "erreurs" du programme ne vont pas être identifiées par **BugAssist** car les contraintes correspondantes ne figurent pas dans le complément d'un sous ensemble de clauses satisfiables de cardinalité maximale.

LocFaults calcule des MCS, c'est à dire le complé-

Programme	Nb_E	Nb_CE	LF	BA
TcasKO	1	131	131	131
TcasKO2	2	67	67	67
TcasKO3	1	23	2	23
TcasKO4	1	20	16	20
TcasKO5	1	10	10	10
TcasKO6	3	12	36	24
TcasKO7	1	36	23	0
TcasKO8	1	1	1	0
TcasKO9	1	7	7	7
TcasKO10	6	14	16	84
TcasKO11	6	14	16	46
TcasKO12	1	70	52	70
TcasKO13	1	4	3	4
TcasKO14	1	50	6	50
TcasKO16	1	70	22	0
TcasKO17	1	35	22	0
TcasKO18	1	29	21	0
TcasKO19	1	19	13	0
TcasKO20	1	18	18	18
TcasKO21	1	16	16	16
TcasKO22	1	11	11	11
TcasKO23	1	41	41	41
TcasKO24	1	7	7	7
TcasKO25	1	3	0	3
TcasKO26	1	11	11	11
TcasKO27	1	10	10	10
TcasKO28	2	75	74	121
TcasKO29	2	18	17	0
TcasKO30	2	57	57	0
TcasKO34	1	77	77	77
TcasKO35	4	75	74	115
TcasKO36	1	122	120	0
TcasKO37	4	94	110	236
TcasKO39	1	3	0	3
TcasKO40	2	122	0	120
TcasKO41	1	20	17	20

TABLE 3 – Nombre d'erreurs localisés pour TCAS

ment d'un sous-ensemble de clauses maximal, c'est à dire auquel on ne peut pas ajouter d'autre clause sans le rendre inconsistant, mais qui n'est pas nécessairement de cardinalité maximale.

BugAssist identifie des instructions suspectes dans l'ensemble du programme alors que **LocFaults** recherche les instructions suspectes sur un seul chemin.

Les travaux futurs concernent à la fois une réflexion sur le traitement des boucles (dans un cadre de bounded-model checking) et l'optimisation de la résolution pour les contraintes numériques. On utilisera aussi les MCS pour calculer d'autres informations, comme le MUS qui apportent une information complémentaire à l'utilisateur.

Remerciements :

Nous tenons à remercier Hiroshi Hosobe, Yahia Lebah, Si-Mohamed Lamraoui et Shin Nakajima pour les échanges fructueux que nous avons eus. Nous tenons aussi à remercier Olivier Ponsini pour son aide et ses précieux conseils lors de la réalisation du prototype de notre système.

Références

- [1] Thomas Ball, Mayur Naik, and Sriram K. Rajamani. From symptom to cause : localizing errors in counterexample traces. In *Proceedings of POPL*, pages 97–105. ACM, 2003.
- [2] Michael Barnett and K. Rustan M. Leino. Weakest-precondition of unstructured programs. In *PASTE'05, ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering*, pages 82–87. ACM, 2005.
- [3] Ilan Beer, Shoham Ben-David, Hana Chockler, Avigail Orni, and Richard J. Treffer. Explaining counterexamples using causality. In *Proceedings of CAV*, volume 5643 of *Lecture Notes in Computer Science*, pages 94–108. Springer, 2009.
- [4] Elazar Birnbaum and Eliezer L. Lozinskii. Consistent subsets of inconsistent systems : structure and behaviour. *J. Exp. Theor. Artif. Intell.*, 15(1) :25–46, 2003.
- [5] John W. Chinneck. Localizing and diagnosing infeasibilities in networks. *INFORMS Journal on Computing*, 8(1) :55–62, 1996.
- [6] John W. Chinneck. Fast heuristics for the maximum feasible subsystem problem. *INFORMS Journal on Computing*, 13(3) :210–223, 2001.
- [7] John W. Chinneck. *Feasibility and Infeasibility in Optimization : Algorithms and Computational Methods*. Springer, 2008.
- [8] Hélène Collavizza, Michel Rueher, and Pascal Van Hentenryck. Cpbpv : a constraint-programming framework for bounded program verification. *Constraints*, 15(2) :238–264, 2010.
- [9] Hélène Collavizza, Nguyen Le Vinh, Olivier Ponsini, Michel Rueher, and Antoine Rollet. Constraint-based bmc : a backjumping strategy. *STTT*, 16(1) :103–121, 2014.
- [10] Alexander Felfernig, Monika Schubert, and Christoph Zehentner. An efficient diagnosis algorithm for inconsistent constraint sets. *AI EDAM*, 26(1) :53–62, 2012.
- [11] Andreas Griesmayer, Roderick Bloem, and Byron Cook. Repair of boolean programs with an application to c. In *Proceedings of CAV*, volume 4144 of *Lecture Notes in Computer Science*, pages 358–371. Springer, 2006.
- [12] Andreas Griesmayer, Stefan Staber, and Roderick Bloem. Automated fault localization for c programs. *Electr. Notes Theor. Comput. Sci.*, 174(4) :95–111, 2007.
- [13] Alex Groce, Sagar Chaki, Daniel Kroening, and Ofer Strichman. Error explanation with distance metrics. *STTT*, 8(3) :229–247, 2006.
- [14] Alex Groce, Daniel Kroening, and Flavio Lerda. Understanding counterexamples with explain. In *Proceedings of CAV*, volume 3114 of *Lecture Notes in Computer Science*, pages 453–456. Springer, 2004.
- [15] James A. Jones and Mary Jean Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *ASE, IEEE/ACM International Conference on Automated Software Engineering*, pages 273–282. ACM, 2005.
- [16] James A. Jones, Mary Jean Harrold, and John T. Stasko. Visualization of test information to assist fault localization. In *ICSE, Proceedings of the 22nd International Conference on Software Engineering*, pages 467–477. ACM, 2002.
- [17] Manu Jose and Rupak Majumdar. Bug-assist : Assisting fault localization in ansi-c programs. In *Proceedings of CAV*, volume 6806 of *Lecture Notes in Computer Science*, pages 504–509. Springer, 2011.
- [18] Manu Jose and Rupak Majumdar. Cause clue clauses : error localization using maximum satisfiability. In *Proceedings of PLDI*, pages 437–446. ACM, 2011.
- [19] Ulrich Junker. Quickxplain : Preferred explanations and relaxations for over-constrained problems. In *Proceedings of AAAI*, pages 167–172. AAAI Press / The MIT Press, 2004.
- [20] Si-Mohamed Lamraoui and Shin Nakajima. A formula-based approach for automatic fault localization in imperative programs. *NII research report*, Submitted For publication, 6 pages, February, 2014.
- [21] Mark H. Liffiton and Ammar Malik. Enumerating infeasibility : Finding multiple muses quickly. In *Proc. of CPAIOR*, volume 7874 of *Lecture Notes in Computer Science*, pages 160–175. Springer, 2013.
- [22] Mark H. Liffiton and Karem A. Sakallah. Algorithms for computing minimal unsatisfiable subsets

- of constraints. J. Autom. Reasoning, 40(1) :1–33, 2008.
- [23] Yongmei Liu and Bing Li. Automated program debugging via multiple predicate switching. In Proceedings of AAAI. AAAI Press, 2010.
 - [24] Joao Marques-Silva, Federico Heras, Mikolás Janota, Alessandro Previti, and Anton Belov. On computing minimal correction subsets. In Proc. of IJCAI. IJCAI/AAAI, 2013.
 - [25] Manos Renieris and Steven P. Reiss. Fault localization with nearest neighbor queries. In Proceedings of ASE, pages 30–39. IEEE Computer Society, 2003.
 - [26] David S. Rosenblum and Elaine J. Weyuker. Lessons learned from a regression testing case study. Empirical Software Engineering, 2(2) :188–191, 1997.
 - [27] Mehrdad Tamiz, Simon J. Mardle, and Dylan F. Jones. Detecting iis in infeasible linear programmes using techniques from goal programming. Computers & OR, 23(2) :113–119, 1996.
 - [28] Xiangyu Zhang, Neelam Gupta, and Rajiv Gupta. Locating faults through automated predicate switching. In Proceedings of ICSE, pages 272–281. ACM, 2006.